

# Код-ревью

Александр Мальцев

18 ноября 2021 г.

# 1 Код на входе

```
20 class UserService {
21     var username;
22     var password;
23
24     constructor(username, password) {
25         this.username = username;
26         this.password = password;
27     }
28
29     get username() {
30         return UserService.username;
31     }
32
33     get password() {
34         throw "You are not allowed to get password";
35     }
36
37     static authenticate_user() {
38         let xhr = new XMLHttpRequest();
39         xhr.open('GET', 'https://examples.com/api/user/authenticate?username=' +
40             UserService.username + '&password=' + UserService.password, true);
41         xhr.responseType = 'json';
42
43         const result = false;
44
45         xhr.onload = function() {
46             if(xhr.status !== '200') {
47                 result = xhr.response;
48             } else {
49                 result = true;
50             }
51         };
52
53         return result;
54     }
55 }
56
57 $('form #login').click(function () {
58     var username = $('#username');
59     var password = $('#password');
60
61     var res = UserService(username, password).authenticate_user();
62
63     if(res == true) {
64         document.location.href = '/home';
65     } else {
66         alert(res.error);
67     }
68 })
```

## 2 Анализ ошибок абстракции

Обычно я бы предпочёл разбить анализ ошибок на категории по сути ошибок: синтаксические, семантические и логические. Однако, в этом участке кода так много проблем, что в этот раз я буду писать плотно замечаний, постепенно улучшая код.

Здесь не так много синтаксических ошибок, сколько семантических и логических. Тем не менее, избавимся от них: Объявление полей класса через `var`, `let`, `const` - некорректный синтаксис. Объявим поля правильно:

```
1  class UserService {  
2      username;  
3      password;  
4  }
```

Использование обычного синтаксиса функций (типа *function()* ) с большой вероятностью будет вести к ошибкам по мере роста кодовой базы. В дальнейшем будем стараться использовать стрелочные функции.

В классе *UserService* мы, во-первых, именуем геттеры точно так же, как и переменные, и, во-вторых, не устанавливаем сеттеры, хотя присваиваем значения полям.

Кроме того, в геттере *username()* мы обращаемся к полю `username` как статическому. Следует заменить *UserService* на *this*.

Я предлагаю написать так:

```

1  class UserService {
2      #_username;
3      #_password;
4
5      constructor(username, password) {
6          this.#username = username;
7          this.#password = password;
8      }
9
10     set #username(value) {
11         this.#_username = value;
12     }
13
14     get username() {
15         return this.#_username;
16     }
17
18     set #password(value) {
19         this.#_password = value;
20     }
21
22     get password() {
23         return this.#_password;
24     }
25 }
26

```

Что касается ошибки, кидаемой геттером *password* (которая, к слову, должна быть не просто строкой, а объектом *Error*) - сейчас я не могу придумать объективной причины запрещать читать пароль юзера, т. к. это не нарушает инвариант объекта и не усложняет пользование его интерфейсом. С таким доводом я делаю геттер для пароля публичным (позже я поясню ещё подробнее).

На данном этапе у нас вырисовывается отдельная сущность, которую было бы неплохо вынести из класса *UserService*.

Так и сделаем - выделим класс *User* как абстракцию для пользователя. В ООП парадигме класс *User* считался бы data-классом

Теперь у нас есть абстракция данных о пользователе *User* и абстракция его сервиса (источник действий над ним) *UserService*. Поскольку мы вынесли сущность, над которой оперировали внутри сервиса, нам те-

перь нужно провести инъекцию зависимости в этот сервис. *Dependency injection* может быть реализован двумя способами:

- Мы оставляем статический метод и передаем в него объект класса *User*
- Мы инициализируем *UserService* объектом класса *User*

Здесь я склоняюсь ко второму варианту. Во-первых, я считаю что статические методы следует расценивать как зло. Во-вторых, мы, скорее всего, захотим в дальнейшем проводить многочисленные операции над юзером посредством *UserService*, поэтому было бы удобно, если бы мы хранили юзера как состояние, к которому имеют доступ все методы сервиса.

Наглядно:

```
27  class UserService {
28      #user;
29
30      constructor(user) {
31          this.#user = user;
32      }
33
34      // Заодно переименуем метод.
35      // И так понятно, кого мы авторизируем
36      authenticate = () => {
37          // TODO: implement
38      }
39  }
40
```

Избавимся от устаревшего *XMLHttpRequest* и заменим его на актуальный *fetch*, рассчитывая, что мы используем транспилятор Babel. Для удобства можем воспользоваться чем-то вроде самописным *URLBuilder* для нашего API (мы бы просто вынесли ответственность за работу с адресами), но эта тема выходит за рамки статьи.

```

27 class UserService {
28     #user;
29
30     constructor(user) {
31         this.#user = user;
32     }
33
34     authenticate = async () => {
35         const url = `https://examples.com/api/user/authenticate`;
36         const headers = {
37             "Content-Type": "application/json",
38         };
39         const body = JSON.stringify({
40             username: this.#user.username,
41             password: this.#user.password,
42         });
43
44         const response = await fetch(url, {
45             method: "POST",
46             headers,
47             body,
48         });
49
50         const result = response.ok ? true : await response.json();
51         return result;
52     };
53 }
54

```

Как мы видим, сервису понадобилось поле *password*. Это причина, по которой мы оставили геттер открытым.

Что касается безопасности - мы теперь отправляем запрос методом POST вместо GET и по протоколу https, что гарантирует нам безопасную доставку данных.

На данный момент код функции может показаться перегруженным. Я согласился бы с этим, в дальнейшем *authenticate* должна подвергнуться рефакторингу (в частности, необходима адекватная абстракция для работы с API).

### 3 Анализ ошибок взаимодействия с интерфейсом

Эта секция о работе с интерфейсом, который мы только что выстраивали.

Здесь я бы мог предложить многочисленные исправления:

1. Безоговорочно переходим с `var` на `let`
2. Убираем *form* из селектора, т. к. селектор по `id` гарантирует нам единственность конкретного элемента на странице (при валидной верстке)
3. Реагируем не на `click`, а на `submit`
4. Пользуемся полем *value* элемента, а не самим элементом (ошибка типов, по сути, следует привлечь Typescript)
5. Создаём новый объект через `new`
6. Обращаемся к полю `location` не у `document`, а у `window`
7. Опционально - пользуемся более семантически корректным `window.location.replace()` вместо `window.location.href`
8. Опционально - пользуемся `FormData`
9. Опционально - отказываемся от JQuery в пользу ванильного Javascript

Получаем что-то вроде этого:

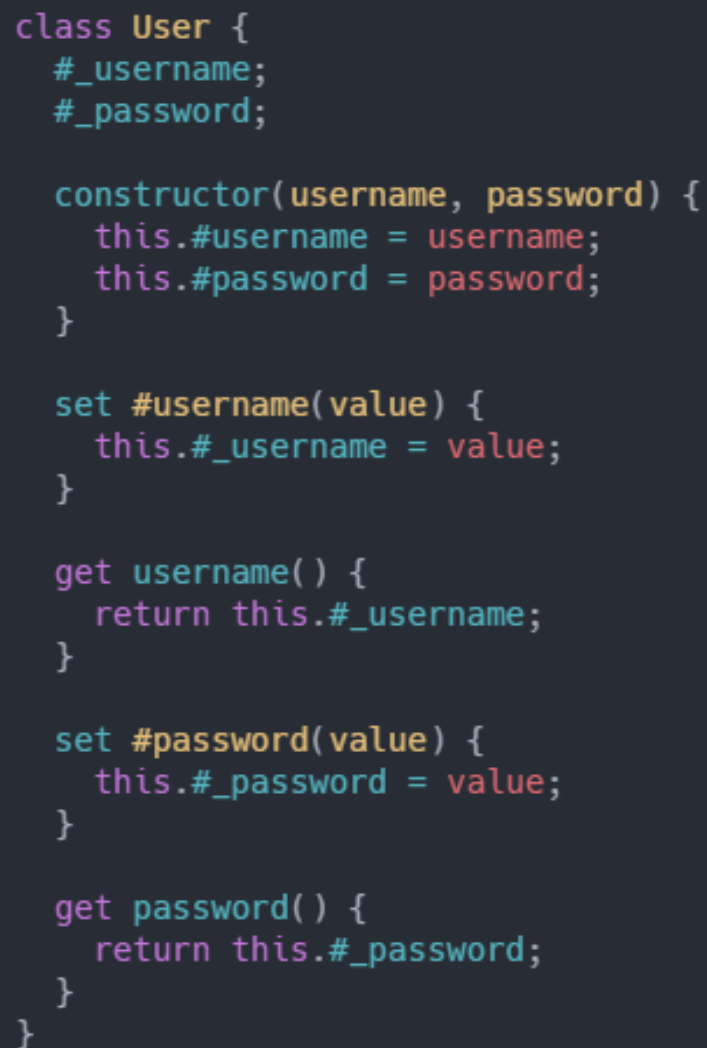
```

55 const loginForm = document.querySelector("#login");
56 loginForm.addEventListener("submit", async () => {
57     const data = new FormData(loginForm);
58     const username = data.get("username");
59     const password = data.get("password");
60
61     const user = new User(username, password);
62     const userService = new UserService(user);
63     const result = await userService.authenticate();
64
65     if (result) {
66         window.location.replace("/home");
67     } else {
68         alert(`Ошибка HTTP: ${result.error}`);
69     }
70 });
71

```



## 4 Итоговый результат



```
class User {  
    #_username;  
    #_password;  
  
    constructor(username, password) {  
        this.#username = username;  
        this.#password = password;  
    }  
  
    set #username(value) {  
        this.#_username = value;  
    }  
  
    get username() {  
        return this.#_username;  
    }  
  
    set #password(value) {  
        this.#_password = value;  
    }  
  
    get password() {  
        return this.#_password;  
    }  
}
```

```
class UserService {
  #user;

  constructor(user) {
    this.#user = user;
  }

  authenticate = async () => {
    const url = `https://examples.com/api/user/authenticate`;
    const headers = {
      "Content-Type": "application/json",
    };
    const body = JSON.stringify({
      username: this.#user.username,
      password: this.#user.password,
    });

    const response = await fetch(url, {
      method: "POST",
      headers,
      body,
    });

    const result = response.ok ? true : await response.json();
    return result;
  };
}
```



```
const loginForm = document.querySelector("#login");
loginForm.addEventListener("submit", async () => {
  const data = new FormData(loginForm);
  const username = data.get("username");
  const password = data.get("password");

  const user = new User(username, password);
  const userService = new UserService(user);
  const result = await userService.authenticate();

  // Обязательно такое сравнение
  if (result === true) {
    window.location.replace("/home");
  } else {
    alert(`Ошибка HTTP: ${result.error}`);
  }
});
```